

Research - Kasteran* — Theorem Proving in Compiler Construction

Alpasan, Lois-Kleinner

2026

Abstract

Theorem proving plays an increasingly vital role in compiler construction, from formal verification of compiler correctness to automated reasoning about program properties. This document surveys the application of SAT/SMT solving, symbolic execution, and interactive theorem proving in compilers, with particular emphasis on the Z3 SMT solver, the SMT-LIB standard, and proof assistants such as Isabelle and Coq. We discuss Kasteran*'s integration of SMT-backed verification for safe code generation and optimization validation.

Kasteran* — Theorem Proving in Compiler Construction

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Theorem proving plays an increasingly vital role in compiler construction, from formal verification of compiler correctness to automated reasoning about program properties. This document surveys the application of SAT/SMT solving, symbolic execution, and interactive theorem proving in compilers, with particular emphasis on the Z3 SMT solver, the SMT-LIB standard, and proof assistants such as Isabelle and Coq. We discuss Kasteran*'s integration of SMT-backed verification for safe code generation and optimization validation.

1. Introduction

Compiler correctness is a fundamental concern: a bug in the compiler can introduce vulnerabilities or incorrect behavior in every program compiled with it. Theorem proving techniques offer a rigorous approach to ensuring compiler correctness, ranging from fully verified compilers (CompCert) to light-weight translation validation. Kasteran* incorporates theorem proving at multiple levels: the type checker uses SMT solving for capability constraint resolution, the optimizer uses symbolic reasoning for transformation validation, and the code generator uses formal models of the target architecture to ensure correct instruction selection.

2. Historical Background

The application of theorem proving to compiler construction began with the pioneering work of McCarthy and Painter on compiler correctness proofs (McCarthy and Painter 1). They proved that a simple compiler for arithmetic expressions produced correct code by establishing a simulation

relation between source language semantics and target machine semantics. This approach—proving that the compiler’s output code correctly implements the source program—established the paradigm for verified compilation.

The development of SAT solving transformed the practical landscape. The DPLL algorithm, introduced by Davis, Putnam, Logemann, and Loveland, provided a complete and efficient procedure for deciding propositional satisfiability (Davis et al. 1). Modern SAT solvers—MiniSAT, Glucose, CryptoMiniSAT—incorporate conflict-driven clause learning (CDCL), lazy data structures, and restart strategies to solve industrial-scale problems with millions of clauses and variables.

SMT (Satisfiability Modulo Theories) solving extends SAT with theory-specific reasoning for arithmetic, arrays, bit-vectors, strings, and data types. The SMT-LIB initiative provides a standard language and benchmark library for SMT solvers (Barrett et al. 1). The Z3 solver from Microsoft Research, developed by Leonardo de Moura and Nikolaj Bjørner, is the most widely used SMT solver in compiler research, supporting a rich set of theories and APIs for integration (de Moura and Bjørner 1).

Interactive theorem proving—exemplified by Coq (developed at INRIA) and Isabelle/HOL (developed at Cambridge and TU Munich)—enables the formal verification of complex software systems. The CompCert verified C compiler, verified in Coq, demonstrates that a realistic optimizing compiler can be proved correct end-to-end (Leroy 1). CompCert uses a series of intermediate languages, each with formal semantics, and proves that each compilation pass preserves semantics.

3. Technical Analysis

The Kasteran* compiler incorporates theorem proving in several phases. The first phase is the capability constraint solver, which operates as an SMT-based type checker. When the type system encounters complex capability constraints (e.g., verifying that fractional capabilities sum to at most 1), the constraint is encoded as an SMT formula and discharged by Z3:

```
// Capability constraint encoding
(declare-fun cap_A () Real)
(declare-fun cap_B () Real)
(declare-fun cap_C () Real)
(assert (and (>= cap_A 0) (<= cap_A 1)))
(assert (and (>= cap_B 0) (<= cap_B 1)))
(assert (and (>= cap_C 0) (<= cap_C 1)))
(assert (= (+ cap_A cap_B cap_C) 1.0))
(assert (> cap_A 0.5)) ; mutable capability required
(check-sat)
```

If the constraints are satisfiable, the type checker produces a capability assignment. If unsatisfiable, the type checker reports an error with a minimal unsatisfiable core, enabling informative error messages.

The second phase is translation validation for optimization passes. After each optimization pass, the compiler encodes the input and output IR as logical formulas and asks the solver to verify equivalence. For SSA-based IRs, equivalence checking reduces to proving that corresponding variables have equal values under all inputs. The compiler generates a verification condition:

```
inputs.semantics_in(inputs) = semantics_out(inputs)
```

For simple optimizations (constant folding, dead code elimination), this check is immediate. For complex transformations (loop vectorization, inlining), the compiler must generate “witnesses”—e.g., mapping functions that relate source loop iterations to target vector iterations (Necula 1).

The third phase is symbolic execution for test case generation. The compiler can symbolically execute a Kasteran* function to enumerate possible execution paths and generate test inputs that achieve high coverage. The symbolic execution engine uses Z3 to solve path constraints, producing concrete inputs for each feasible path (Cadaru et al. 1).

4. Current State of the Art

The field of verified compilation has advanced significantly beyond CompCert. CakeML, a verified ML implementation, proves correctness of bootstrapping compilation from a functional language to machine code (Myreen et al. 1). The Vellvm project provides a Coq formalization of LLVM IR, enabling verification of LLVM transformation passes (Zhao et al. 1). Alive, a domain-specific language for LLVM peephole optimizations, uses SMT solving to automatically verify optimization correctness (Lopes et al. 1).

The use of SMT in compilers has become standard practice. LLVM’s NewGVN pass uses Z3 for querying equivalence of values. GCC’s EVRP pass uses a simple solver for range analysis. The Emla compiler for the HomeBrew platform uses SMT solving for instruction selection (Singer and French 1).

Dafny, a programming language with built-in verification capabilities, demonstrates the integration of theorem proving into a practical programming environment (Leino 1). Dafny’s type system includes preconditions, postconditions, loop invariants, and termination metrics, which are verified using Boogie and Z3.

5. Relevance to Kasteran*

Kasteran*’s theorem proving integration is designed to be transparent to the programmer for common cases while providing powerful verification capabilities when needed. The compiler automatically generates verification conditions for: (1) capability constraint satisfaction, (2) memory safety of unsafe blocks, (3) termination of recursive functions, (4) absence of integer overflow, and (5) compliance with effect annotations.

The programmer can annotate functions with preconditions and postconditions, which are verified at compile time:

```
fn divide(a: Int, b: Int) → Int
  requires b  0
  ensures result  overflow
{
  a / b
}
```

The compiler encodes these contracts as SMT queries, ensuring that callers satisfy preconditions and callees satisfy postconditions.

6. Future Directions

The integration of theorem proving and compilation continues to advance along several fronts. The use of machine learning to guide SMT solving—predicting which lemmas to instantiate, which theory combination strategies are effective—promises to make automated reasoning more efficient. The development of certified compilation pipelines for domain-specific languages—where high-level DSLs are compiled to verified low-level implementations via proved-correct translations—remains an active research area.

Another frontier is the verification of compiler optimizations, not just correct compilation. While CompCert proves that semantics are preserved, it does not prove that the generated code is optimal. Formalizing optimality criteria and proving that a compiler achieves them is significantly more challenging.

Works Cited

- Barrett, Clark, et al. “SMT-LIB: The Satisfiability Modulo Theories Library.” *Proceedings of the 4th International Conference on Theory and Applications of Satisfiability Testing*, 2004, pp. 1-7.
- Cadar, Cristian, et al. “EXE: Automatically Generating Inputs of Death.” *ACM Transactions on Information and System Security*, vol. 12, no. 2, 2008, pp. 1-38.
- Davis, Martin, et al. “A Machine Program for Theorem Proving.” *Communications of the ACM*, vol. 5, no. 7, 1962, pp. 394-397.
- de Moura, Leonardo, and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337-340.
- Leino, K. Rustan M. “Dafny: An Automatic Program Verifier for Functional Correctness.” *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2010, pp. 348-370.
- Leroy, Xavier. “Formal Verification of a Realistic Compiler.” *Communications of the ACM*, vol. 52, no. 7, 2009, pp. 107-115.
- Lopes, Nuno P., et al. “Alive: A Domain-Specific Language for Automatic Verification of LLVM Optimizations.” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 1-12.
- McCarthy, John, and James Painter. “Correctness of a Compiler for Arithmetic Expressions.” *Mathematical Aspects of Computer Science*, vol. 19, 1967, pp. 33-41.
- Myreen, Magnus O., et al. “A Verified ML Compiler and Its Machine-Code Semantics.” *Journal of Automated Reasoning*, vol. 53, no. 3, 2014, pp. 213-247.
- Necula, George C. “Translation Validation for an Optimizing Compiler.” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 83-94.
- Singer, Jeremy, and Tim French. “SMT-Based Instruction Selection for Embedded Compilers.” *Proceedings of the 2018 International Conference on Compiler Construction*, 2018, pp. 1-11.
- Zhao, Jianzhou, et al. “A Formal Semantics of LLVM IR in Coq.” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 1-12.

- Bertot, Yves, and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- Nipkow, Tobias, et al. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- Pnueli, Amir, et al. “Translation Validation: From DC+ to C.” *Proceedings of the 1998 International Conference on Computer Aided Verification*, 1998, pp. 1-15.
- Clarke, Edmund M., et al. “Model Checking and Abstraction.” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, 1994, pp. 1512-1542.
- Godefroid, Patrice, et al. “Automated Whitebox Fuzz Testing.” *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008, pp. 1-16.
- Tillmann, Nikolai, and Wolfram Schulte. “Parameterized Unit Tests.” *Proceedings of the 10th European Software Engineering Conference*, 2005, pp. 253-262.
- Beyer, Dirk, et al. “The Software Verification Competition (SV-COMP).” *Proceedings of the 2019 International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019, pp. 1-20.
- Gurfinkel, Arie, and Sharon Shoham. “Abstraction and Refinement in SMT-Based Software Verification.” *Proceedings of the 2015 International Conference on Verification, Model Checking, and Abstract Interpretation*, 2015, pp. 1-19.
- Jaffar, Joxan, et al. “A Complete and Efficient SMT Solver for Polynomial Constraints.” *Proceedings of the 2014 International Conference on Computer Aided Verification*, 2014, pp. 1-17.
- Claessen, Koen, and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.” *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 268-279.